Motivation and Framework for Using Genetic Algorithms for Microcode Compaction

Steven Beaty
Department of Mechanical Engineering

Darrell Whitley
Department of Computer Science

Gearold Johnson
Department of Mechanical Engineering
Colorado State University
Fort Collins, Colorado, 80523

Abstract

Genetic algorithms are a robust adaptive optimization technique based on a biological paradigm. They perform efficient search on poorly-defined spaces by maintaining an ordered pool of strings that represent regions in the search space. New strings are produced from existing strings using the genetic-based operators of recombination and mutation. Combining these operators with natural selection results in the efficient use of hyperplane information found in the problem to guide the search. The searches are not greatly influenced by local optima or non-continuous functions. Genetic algorithms have been successfully used in problems such as the traveling salesperson and scheduling job shops. Microcode compaction can be modeled as these same types of problems, which motivates the application of genetic algorithms in this domain.

1 Introduction

Microcode compaction techniques have been based on heuristics thought to produce the shortest final code sequences. In Landskov et al. [LDSM80] these are characterized into the following types: linear analysis, critical path, branch and bound, and list scheduling. The approach put forth in this paper does not have its basis in heuristics nor does it fall within the proper bounds of the previous classifications. The method proposed uses genetic algorithms, an artificial intelligence technique, to correctly schedule micro-operations into micro-instructions.

Because the cost of developing micro-architectures is great, compaction should make maximal usage of the resources available in a machine. This goal is complicated by the fact that, as Robertson [Rob79] has shown, the entire microcode compaction process is NP-complete. To generate code in a reasonable amount of time, previous methods have reduced the search space of the problem by introducing heuristics that remove areas that do not appear to benefit the final code sequence. This is a reasonable approach given the nature of the algorithms used to search for correct solutions.

Another approach would be to attempt to increase the effectiveness of the search techniques. Genetic algorithms have been already been applied to known difficult search problems. Good results have been obtained on the well-known traveling salesperson (TSP) and job shop scheduling problems. Both problems optimize a graph representation that includes constraints. The TSP constrains the order that nodes can be visited such that all nodes must be visited once and only once. Job shop scheduling has both resource conflicts and timing constraint between nodes. Microcode compaction has similar constraints in the form of a data dependency graph. The application of genetic algorithms, known to be good at optimizing graph-based problems, to compaction would seem appropriate.

An environment, such as the compiler described by Beaty et al. [BDM+88], is assumed for this work. Such an environment needs to contain a machine description that includes all information relevant to compaction including

- machine-level resource descriptions with timing constraints,
- field encoding format and field requirements for each individual machine operation, and
- dataflow information for each operation.

The environment must also produce a data structure amenable to the compactor. This structure usually takes the form of a graph that represents both the operations to be performed and any constraints between successive operations. Any milieu producing these requirements (such

as stand-alone assemblers or optimizers) is sufficient; a retargetable, optimizing, microcode compiler is the basis for this work.

2 The Microcode Compaction Problem

The problem that compaction tries to solve is placing a group of micro-operations (MOs) necessary for correct execution of the algorithm into as few micro-instructions (MIs) as possible. An optimal answer has been shown to be NP-complete, i.e. computationally expensive. There are many problems related to compaction such as the TSP and processor scheduling problem but Landskov et al. [LDSM80] states compaction is an example of job shop scheduling, giving a direction of attack.

2.1 Representation

Most compaction algorithms run on data dependency graphs (DDGs). A DDG is composed of nodes that represent MOs and arcs that enforce a partial ordering on the nodes. The nodes contain information that constrain their placement with other nodes, such as resource and instruction field information. Machine resource information, in our environment, is represented as live tracks. This information is read from a file that contains all machine specific details. A live track contains the name of the resource and whether the resource is live in, live out or defined in this node. Nodes may have multiple live tracks for any resource, indicating repeated usage of the resource. Field usage for a node is maintained in a bit vector form for rapid comparison.

The arcs are directed such that an arc from node X to node Y implies that MO Y cannot be placed in an MI higher than the MI in which MO X is placed. That is, Y cannot execute in the machine before X and preserve the semantics of the algorithmic description. There is timing information placed along the arcs specifying how soon after the preceding node a successor node may be executed. This is data dependency information, i.e. how long the value in a required resource is valid. This timing is specified to have a minimum and a maximum value. So, for example, if the arc between X and Y had a minimum time of 2 and a maximum time of 6, Y cannot be placed in an MI until 2 time units after X and must be placed before 6 units have expired. So the information written by X to a resource that Y reads is available for four specific cycles in the machine. The time units can be based on any convenient measure such as major or minor cycles in the target architecture.

Data anti-dependencies describe the amount of time after a resource has been read that it may be rewritten. Vegdahl [Veg82] noted that more attention should be given to the ordering of data anti-dependencies because of the large effect on both resulting code and execution time of the compactor.

Given a DDG that represents the action to be taken in the target machine, the compaction algorithm attempts to place all nodes in as few MIs as possible given the following constraints:

- data dependencies in the graph are maintained,
- resource usage cannot conflict within an MI,
- encoding fields cannot conflict within an MI.

2.2 Previous Methods

Landskov et al. [LDSM80] elucidates four methods of microcode compaction:

- linear analysis
- critical path
- branch and bound
- list scheduling.

Linear analysis attempts to place an MO in the highest MI possible. When an MO does not fit within an existing MI, a new one is formed to contain it. Critical path finds the shortest viable path through the MOs and places them in the required number of MIs first. It then attempts to place all remaining MOs into those MIs. Failing that, it creates new MIs to receive any leftover MOs. Branch and bound forms trees of possible MIs; a new branch is formed wherever more than one MI could be placed at that point in the list. List scheduling starts with an initially empty MI list. MOs are placed within this list when they are

1. data ready (that is, when all the resources they depend upon have the proper values), and
2. highest on the data ready list as judged by a weighting heuristic.

These all are methods for intrablock compaction.

Recent efforts have focused on interblock compaction as well. Trace scheduling and percolation scheduling are two methods. Trace scheduling extends local compaction methods by allowing them to operate on more than one block at a time [Fis81, HMS87]. A trace is a loop-free section of code that contains multiple basic blocks. When an entire trace is scheduled, the possibility for creating

incorrect code arises. A bookkeeping phase makes copies of operations along other traces to correct the semantics. Traces are chosen in order of most-likely-to-execute first. Other traces can then be compacted. This emphasizes the frequently traversed paths at the possible expense of the infrequently traversed ones.

Percolation scheduling, as described by Nicolau [Nic85], has a small set of operations allowed between MIs in a program graph. These allow deletion of empty MIs, non-flow control MO motion, movement of a conditional jump, and removal of redundant MOs. The operations are defined so that the intended semantics are always preserved. Inverses of these functions are also allowed. The application of the operators is guided by higher level heuristics.

2.3 Heuristics

All methods so far rely on heuristics to remove parts of the search space that appear fruitless. Linear analysis tries to place MOs in the highest possible MI so that it reduces interference with other MOs that need to be placed. It also uses a first-come first-served ordering on the MOs to be placed. The critical path method places MOs not on the critical path heuristically. Branch and bound reduces the building and searching of the trees by guessing which branches will not lead to more compact code. Packing of MOs into MIs are ranked according to some metric. List scheduling has a similar approach in that MO placement is based on a set of weights assigned to each MO. The MO that is placed next in the MI list is the one on the data ready list with the highest weight. Some of the various criteria for weighting as suggested by Allan [All86] are

- difference between current MI and the position this MO must be placed,
- average of non-infinite successor arc lengths, allowing more tightly constrained nodes to be placed first,
- number of non-infinite successor arcs, and
- height in uncompacted DDG.

A combination of these and other weights can be used to drive the list scheduling algorithm. A polynomial used for varying the priority of the different weights on a per machine basis was also implemented in Allan [All86].

The use of heuristics can be difficult when trying to arrive at an efficient yet efficacious compactor. This difficulty is compounded by several factors. The heuristics generally must be regenerated for each machine targeted. The heuristics themselves are not in a form easily understood by humans. This makes it difficult for humans to correctly guess and modify a compactor's behavior. It is also possible that the heuristics do not address an issue that has great import on the final code. Heuristics that work well for one ordering of MOs may not work well for another.

3 Genetic Algorithms

Genetic algorithms (GAs) manipulate populations of strings that represent the parameterization of the optimization problem. The strings correspond to chromosomes or genotypes in biological terms. There is a mapping from this representation to the phenotype of the actual solution. GAs use a form of selective pressure to encourage over-achieving and discourage under-achieving strings in the population. A string's chances of reproducing correspond to its performance in the current environment. This is an easily understandable method and it produces robust searches of difficult parameter spaces as demonstrated by Holland and others [Hol75, DeJ86, Gol89].

3.1 Foundations

Parameters are usually encoded into some form of binary representation. This representation is then used for subsequent operations and evaluations. Consider the string: `1100101001110110001`. This could represent an integer, a fixed or floating point real, or any other relevant model of the parameters to be optimized. Multiple parameters are simply appended together. The initial population is usually generated by creating random strings.

To perform recombination, the basis for most genetic adaptation in nature, consider also the string: `xyyxxyxyxyxxxyyyxxy` (with `x` for `0` and `y` for `1`) and some number of break points. The genetic material from one string is then swapped between those break points with the corresponding material from the other. An example with two break points is:

```
11001 \/ 01001110110 \/ 001
xyyxx /\ yxyxyxxxyyy /\ xxy
```

resulting in the two children:

```
11001yxyxyxxxyyy001
xyyxx01001110110xxy
```

Although a single break point is usually used in discussions of GAs, two have been empirically shown by Booker [Boo87] to produce better results.

Another operation in the reformation of strings is mutation. This is accomplished by randomly toggling some of the bits in the offspring. This creates genetic diversity. It has been found, in the general case, that mutation rates should be kept low (less the 5%) for best exploitation and least disruption of the information present.

In standard GAs, all the strings in the population are reformed during a generation. Parents are crossed on the basis of their performance in comparison to the average fitness of the population and mutation is allowed to occur on the offspring. The selective pressure is provided by the fitness measure; the differential need not be great to achieve good results. Both selective pressure and initial population sizes may be tuned to match the problem space. The type of crossover and rate of mutation needs to be selected based on the problem type.

To relate the encoding with the sampling of hyperspace, consider a string of length three. With this we get the ability to represent a three-dimensional hypercube. The string `011` represents a corner of the hypercube. Edges have one of the bits as a "don't care", i.e. `01*`. Faces have two "don't cares": i.e. `0**`. The entire space can be expressed by a complete "don't care string": i.e. `***`. Strings that contain a "don't care" in some position are termed schemata. In general, each binary encoding corresponds to one corner in the hypercube and samples $2^L - 1$ different hyperplanes in the search space where L is the length of the binary encoding. This is the idea of "intrinsic parallelism" whereby one string samples the productivity of many hyperplanes [Hol75]. The schema theory indicates that individual hyperplanes will increase or decrease their representation in a population based upon their relative fitness in that population when reproduction and recombination are applied.

The more diverse the original population, the more global the search. The search does not avoid or escape from local minima; it does a global search where local minima are ignored in favor of higher-valued strings. If a local minima is found to be best, it will tend to be competitive with all areas of the space searched. It has been shown that if an area in hyperspace has above average performance and is sampled by a schema in the population, that area's representation will increase within the population. It has been calculated that for the processing of N structures per generation approximately N^3 schemata are sampled (intrinsic parallelism).

The ability to sustain search is dependent upon the genetic diversity in the population. When a population lacks diversity, new areas of the space are not examined. Mutation can be used to drive the search into these unexamined areas. However, a fixed level of mutation has been shown to disrupt the search early and then fail to provide enough diversity in the later stages. Thus, adaptive mutation increases the mutation rate based on the homogeneity of the population and gives better performance.

3.2 GENITOR

The GENITOR GA program, developed by Whitley [WSF89, WSS90], has some differences with "standard" GAs that appear to increase performance. It does not replace the entire population with each generation. Instead it probabilistically chooses two parents to reform into two offspring. Recombination and mutation occur, then one of the offspring is discarded randomly. The remaining offspring is placed in the population according to its fitness in relation to the rest of the strings. The lowest-valued string is discarded. This keeps high-valued strings within the population, directly accumulating high-performance hyperplanes. It also bases the reproductive opportunity upon rank with the population, not upon a string's fitness value in comparison with the average of the population, reducing the impact of selective pressure fluctuation. It also reduces the importance of choosing a proper evaluation function for fitness in that the difference in the fitness function between two adjacent strings is irrelevant.

A recent improvement has been made to GENITOR in the form of a distributed genetic search. This is not simply running subpopulations on different processors but also occasionally swapping the best members from neighboring subpopulations. In this way, subpopulations are somewhat independent while still sharing some information. The search speed is increased over the serial method, allowing larger populations to be explored in a given amount of time. Larger problems are also approachable with the distributed genetic search. It is also a more robust method, producing better results without as much sensitivity to population size or selective pressure. The improvements are attributed to the maintenance of genetic diversity by the interacting subpopulations.

4 Results from Related Work

Realistic scheduling problems are difficult to define using traditional mathematical techniques. As a result, more traditional optimization methods are difficult to apply. GAs are capable of searching ill-structured spaces and also provide a global method of search.

Genetic algorithms have been recently applied to two areas of interest with good result: the traveling salesperson problem and job shop scheduling. The following two subsections discuss the motivation and results reported by Whitley and others [WSF89, WSS90, CS89]. High quality solutions have been found for both problems. The results are not based upon heuristics or local optimization information. As is common with GAs, a method of ranking the current population is required. In both these problems, this is a simple task of summation to find the length of each

member of the population. This task is, if anything, easier for microcode compaction in that the length of compacted code is trivial to find.

4.1 The Traveling Salesperson Problem

This problem involves finding the shortest Hamiltonian path or cycle in a graph where nodes represent cities, and edges represent the paths and distances between two cities. The optimal solution is one that has the least distance and yet visits all the nodes (cities). The TSP is an example of a problem that is NP-hard; all known methods for finding an optimal solution require searching a space that grows exponentially with the number of nodes in the graph.

When using GAs to perform TSP optimization, what is desired is to maintain any good subtours present in the parents. This leads to shorter overall tours in the children. A recombination operator that preserves edges will exploit the most amount of information from the parents. In Whitley et al. [WSF89, WSS90] this is achieved by making an edge map and having the recombination operator use this map. It is possible to show that this method changes the sampling rate of hyperplanes in the N-dimensional hypercube (where N is the length of the encoding of the problem) in favor of high fitness hyperplanes. This method of optimization does not use any information on the distance between cities, only the distance of the overall tour. The use of this metric for the evaluation function is important because of its simplicity and applicability to many forms of scheduling and sequencing including the microcode compaction problem.

The published results from this method are impressive. The best known solution for a certain 30-city TSP problem is 420. With a serial version of GENITOR, a population of 200 and allowing up to 70,000 recombinations, the GA found the best known solution in 28 of 30 tries. It found a solution of 421 with the other two tries. With the distributed version, it found the best known solution in 30 of 30 attempts. On a 105-city problem, with the distributed version and 10 subpopulation of 1000 each and allowing for 200,000 recombinations in each subpopulation, the best known value of 14,383 was found 15 out of the 30 times. In the remaining 50% of the time, the solutions were within 1% of the best known. More recent enhancements to the algorithm have further improved performance (solutions of 420 on 45 of 45 tries) while reducing search time by more than 50(a maximum of 30,000 recombinations).

4.2 Job Shop Scheduling

In scheduling machine usage on a job shop floor, the flexibility is usually found in the sequence of jobs presented to the line. There are fixed setup, idle, and active costs for machines. A strict amount of product needs to be produced in order to meet the demand. Therefore, the approach taken is viewing the optimization problem as one of sequencing the types of jobs presented to the first machine in the line. This can then be viewed as a problem similar to the TSP. The sequence is then evaluated on the basis of total cost and the GA performs the search accordingly.

In Whitley et al. [WSS90] a detailed description of an actual production line in use at Hewlett-Packard in Fort Collins, Colorado is discussed. It contains 6 workcells (groups of machines) in sequence, each performing a specific operation. Each has a single input and a single output queue. Every workcell contains two identical machines operating independently. The machines have costs associated with the various tasks they perform. There are twenty different types of products that are produced by the line.

Two different approaches were tried: 1) a strict FIFO where the GA controlled the sequence of jobs presented to the line, and 2) a HYBRID where the GA attempted to optimize the initial sequence and a greedy algorithm attempted to reorder jobs in the line for maximal machine usage. The FIFO job appears to be more difficult because it is not allowed to reorder jobs within the line. Both models try to keep all machines busy all the time. Surprisingly, the FIFO model produced better sequences, i.e. kept more machines busy more of the time resulting in lower cost. It also produced results faster than the HYBRID method. The results were not greatly different (approximately 3%) but the implication of not having to use a greedy, heuristic-based method (requiring more code and effort) are great. It is thought that FIFO worked better than HYBRID because of its ability to directly control all the global information. What appears good from a local greedy point of view is not always good from a global perspective. The FIFO is also probably a more realistic model of many scheduling tasks.

The implications of using only overall cost for the evaluation function are also significant. This allows for the application of GAs to other scheduling or sequencing problems that are not well-structured enough for traditional optimization methods. The only two areas that vary between problems are

1. the representation scheme and
2. the evaluation function.

It has been shown empirically that these are easy to vary.

[WSS90] concludes "that one should not attempt to build heuristics into the scheduling system, but rather let the genetic algorithm do the work." By doing this, the implementation is simplified, the computational complexity is reduced and the quality of the results are not damaged.

5 Framework for Scheduling Data-Dependency Graphs

This section describes a framework for transforming DDGs into scheduled microprograms. It must be emphasized that this is only one of the plethora of possible methods. For all methods in this vein there are several design components that must be chosen. These are 1) the syntax of the chromosomes, 2) the interpretation of the chromosomes, and 3) the set of operators that transform the chromosomes. Other controlling parameters may be tuned to enhance the performance of the GA. These include population size, parent selection method, choice of evaluation function, and mutation rate. These have less impact upon the GA than do the chromosome operators so less attention will be paid to them.

One of the chromosome operators is mutation. Mutation simultaneously increases genetic diversity and broadens the search space. The method of mutation chosen moves an MO from one MI to another. This is allowed only when no code movement constraints are broken. Empty MIs are not removed from the list during compaction as they add scheduling flexibility. In the final schedule, all non-essential empty MIs are removed. If a mutation-only approach is effective, this implies that

1. there are few local minima, and similarly
2. there are many good solutions.

Our assumption is that these two conditions are not present in compaction as a rule so the crossover operator becomes important.

To examine the crossover operator, a relevant example proves beneficial. The following is an overview of a complex schedule optimization problem to which Syswerda [Sys90] applied GAs. The problem involves scheduling machinery in a lab cognizant of

- resource constraints between simultaneous users,
- time constraints within the workday,
- setup time for each task,
- priority of each task, and
- precedence in the ordering of tasks.

Information required to codify the process of creating efficient schedules is difficult to obtain, because of its complex nature and the difficulty of expressing it in a succinct rule-based form. GAs do not require this codification, an example of domain-specific information.

The scheduling problem is one of placing tasks in a matrix of resources versus time. Contention for a resource is not allowed at any time during the schedule. If resource overlap is empty, multiple tasks may be executed simultaneously. A complete schedule contains possibly overlapping tasks ordered in an ascending time sequence. Time is discrete with tasks allowed to start only on an hour boundary and to use only integral hour increments. Using this approach, the problem may be viewed as either an ordering or a combinatoric one. The list of tasks must be placed in a particular order; not all orders are legal because they might violate the constraints mentioned above.

To achieve legal orderings a deterministic schedule builder is used. This receives an ordered list of tasks and builds a schedule based upon that. It places tasks into the matrix by choosing from the front of the ordered list. This scheduler guarantees that all constraints hold for all tasks placed. The process continues until all tasks are scheduled. This process is very similar to microcode list scheduling except heuristics are not used to choose a task's priority. What determines a task's priority is its position within the list: it is this ordering that the GA optimizes. This has the benefit that the GA does not have to know anything about the specific scheduling task. It only has to perform perturbations on the priority list. Adjacency is unimportant within the list; resource contention is resolved simply by position.

The schedule builder used in [Sys90] is not very complex, allowing and demanding the GA to produce good orderings. This provides the GA with, and control of, all information pertinent to the scheduling process. The evaluation function (based on how well priority jobs are placed within the schedule) provides sufficient feedback.

Three different mutation and three different crossover operators are compared to a simple random search. Each of the mutation operators produced results much better than simple random search. The best mutation operator simply switched two tasks selected randomly. Each crossover operator also produced results better than random search. In subsequent tests, each crossover operator was combined with the order-based mutation operator, and these combinations produced the best overall results. It appears that crossover plays a more significant role early in the generations with mutation playing a more important role as populations converge.

The similarities between this problem description and solution and those of list scheduling microcode are plain. The encouraging results in both this system and others point to GAs application to the compaction problem. The simple schedulers and domain-independent optimizations promise easy and effective compactors. While much needs to be done to validate this approach, the exploration should prove fruitful.

6 Conclusion

Genetic algorithms provide for machine-independent microcode compaction optimizations. Their performance on closely related problems suggest that they are well-suited for the task. GAs remove the need to specify good heuristics by using only the total length of the resulting microcode as a metric. This allows more of the search space to be visited, reducing the effects of local minima. They will combine two well-suited graphs to form another, preserving areas of well-compacted microcode in the process. The operators are simple and easy for a human to comprehend.

A possible drawback is the time required by using GAs to perform compaction. This will certainly be greater than most of the heuristic-based methods. It should be noted, however, that the time taken by a GA-based approach is eminently controllable. One only has to limit the size of the population and number of recombinations to control the total time. There exists a classic tradeoff between running time and other benefits. GAs should be considered when ease of retargetability, lack of heuristics, and quality of produced code are significant concerns.

References

[All86] V.H. Allan. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[BDM+88] S.J. Beaty, M.R. Duda, R.A. Mueller, P.H. Sweany, and J Varghese. "Optimization issues for a retargetable microcode compiler". *IEEE MicroArch*, 3(1):5–15, December 1988.

[Boo87] L. Booker. "Improving search in genetic algorithms". In Lawrence Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 61–73. Morgan Kaufmann, 1987.

[CS89] Gary A. Cleveland and Stephen F. Smith. "Using genetic algorithms to schedule flow shop releases". In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[DeJ86] K. DeJong. *An Analysis of Reproduction and Crossover in a Binary - coded Genetic Algorithm*. PhD thesis, University of Michigan, Ann Arbor, 1986.

[Fis81] J.A. Fisher. "Trace scheduling: A technique for global microcode compaction". *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[Gol89] David Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[HMS87] M.A. Howland, R.A. Mueller, and P.H. Sweany. "Trace scheduling optimization in a retargetable microcode compiler". In *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, Colorado Springs, CO, December 1987.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. "Local Microcode Compaction Techniques". *ACM Computing Surveys*, 12(3):261–294, September 1980.

[Nic85] Alexandru Nicolau. "Percolation scheduling: A parallel compilation technique". Technical report, Department of Computer Science, Cornell University, Ithaca, New York, May 1985.

[Rob79] E.L. Robertson. "Microcode Bit Optimization is NP-complete". *IEEE Transactions on Computers*, C-28(4):316–319, April 1979.

[Sys90] Gilbert Syswerda. "Schedule optimization using genetic algorithms". In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.

[Veg82] S.R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[WSF89] D. Whitley, T. Starkweather, and D. Fuquay. "Scheduling problems and traveling salemen: The genetic edge recombination operator". In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[WSS90] D. Whitley, T. Starkweather, and D. Shaner. "The traveling saleman and sequence scheduling quality solution using genetic edge recombination". In L. Davis, editor, *The Genetic Algorithms Handbook*. 1990.